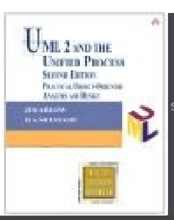
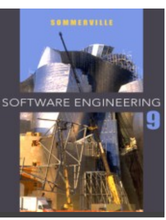
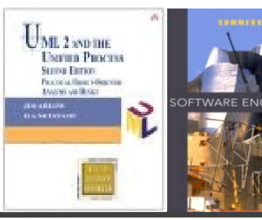



Object Oriented Analysis

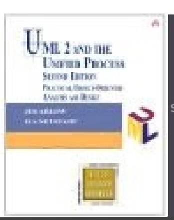
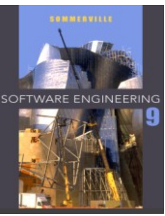
Lecture 5

Outline



- ✧ Objects and classes [Lecture 4]
- ✧ Finding analysis classes [Lecture 4]
- ✧ Relationships between objects and classes
  - Links
  - Associations
  - Dependencies
- ✧ Inheritance and polymorphism
- ✧ Interaction diagrams

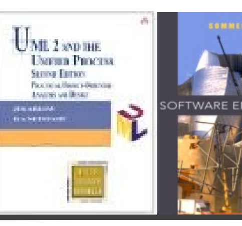
© Clear View Training 2010 v2.6

Relationships Between Objects and Classes

Lecture 5/Part 1

What is a relationship?



- A *relationship* is a connection between modelling elements
- In this section we'll look at:
  - *Links* between objects
  - *Associations* between classes
    - aggregation
    - composition
    - association classes
  - *Dependencies* between model elements

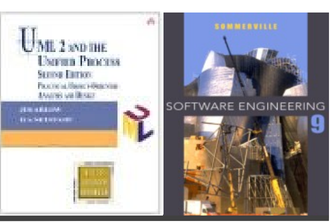
© Clear View Training 2010 v2.6

What is a link?

- Links are connections between objects
  - Think of a link as a telephone line connecting you and a friend. You can send messages back and forth using this link
- Links are the way that objects communicate
  - Objects send messages to each other via links
  - Messages invoke operations
- OO programming languages implement links as object references or pointers. These are unique handles that refer to specific objects
  - When an object has a reference to another object, we say that there is a *link* between the objects

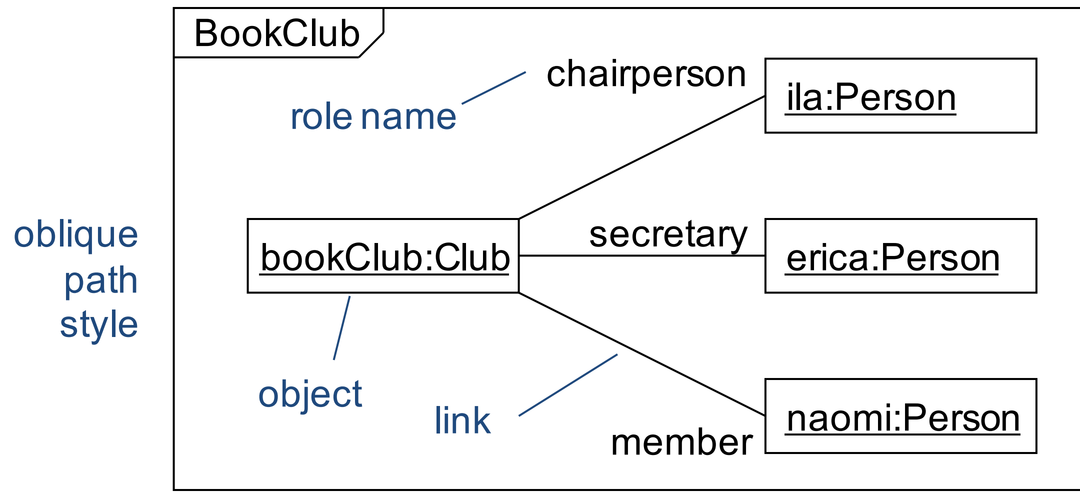
© Clear View Training 2010 v2.6

Object diagrams

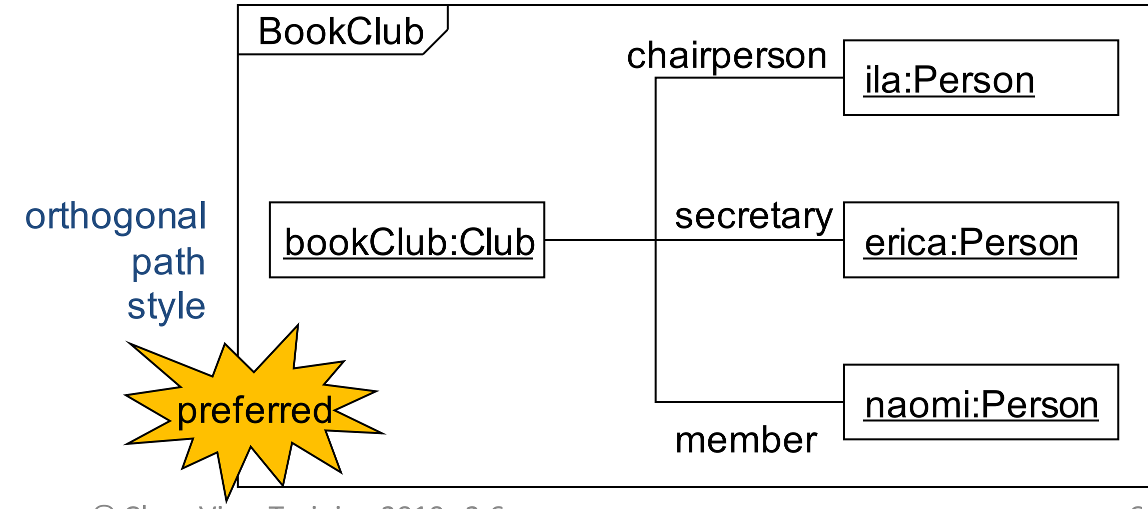


- Paths in UML diagrams (lines to you and me!) can be drawn as orthogonal, oblique or curved lines
- We can combine paths into a tree *if* each path has the same properties

oblique path style

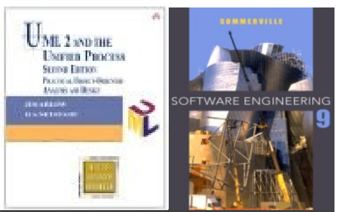


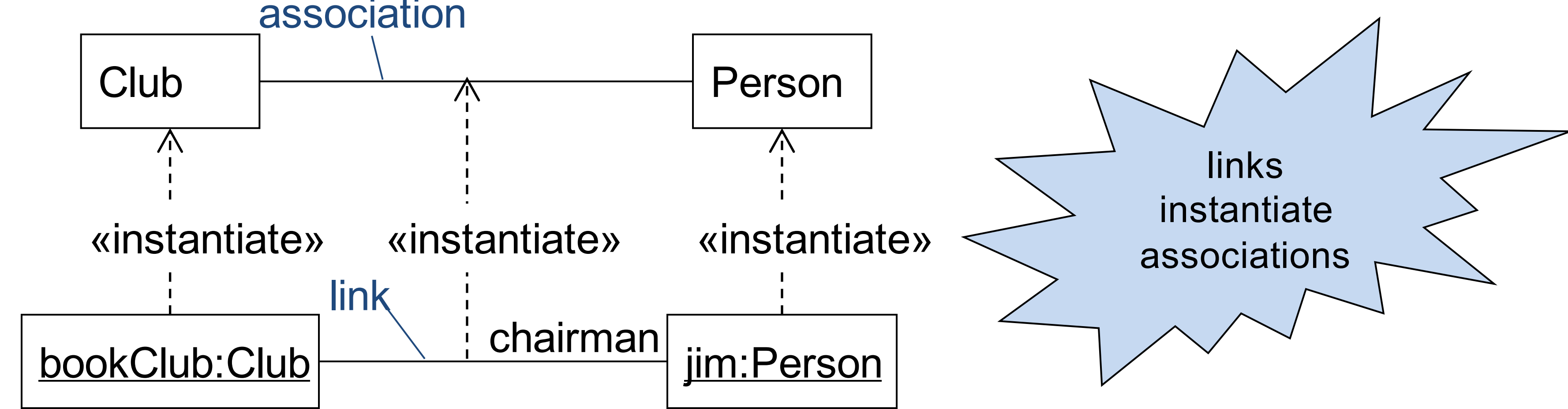
orthogonal path style



© Clear View Training 2010 v2.6

What is an association?

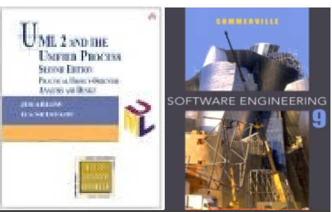


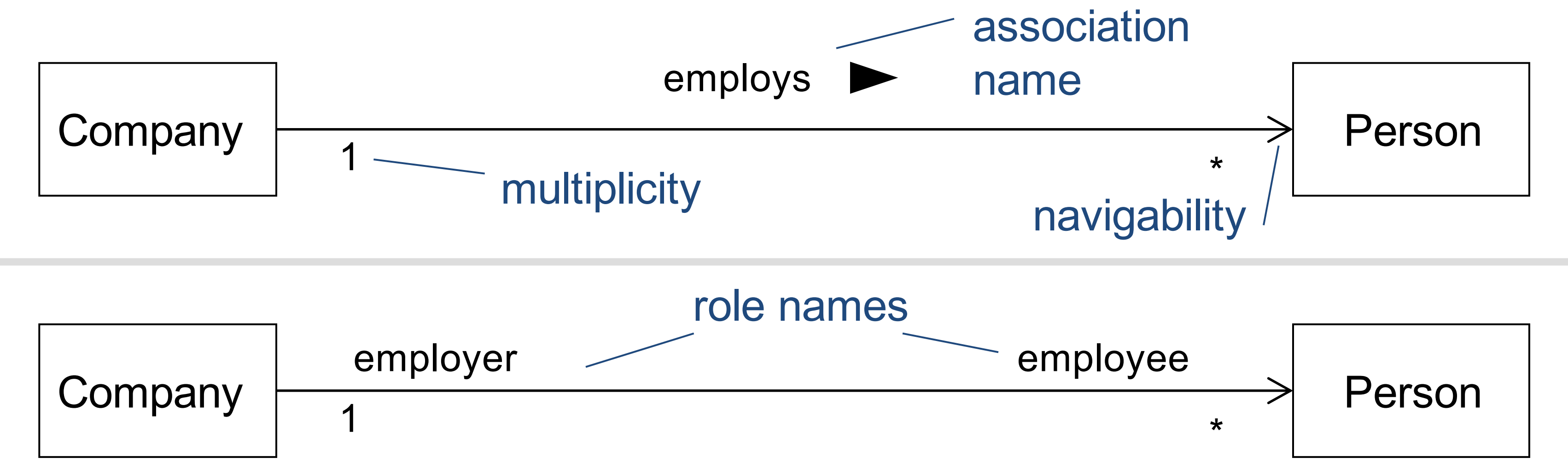


- Associations are relationships between classes
- Associations between classes indicate that there are links between objects of those classes
- A link is an instantiation of an association just as an object is an instantiation of a class

© Clear View Training 2010 v2.6

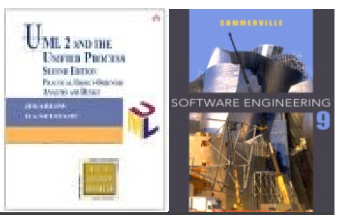
Association syntax

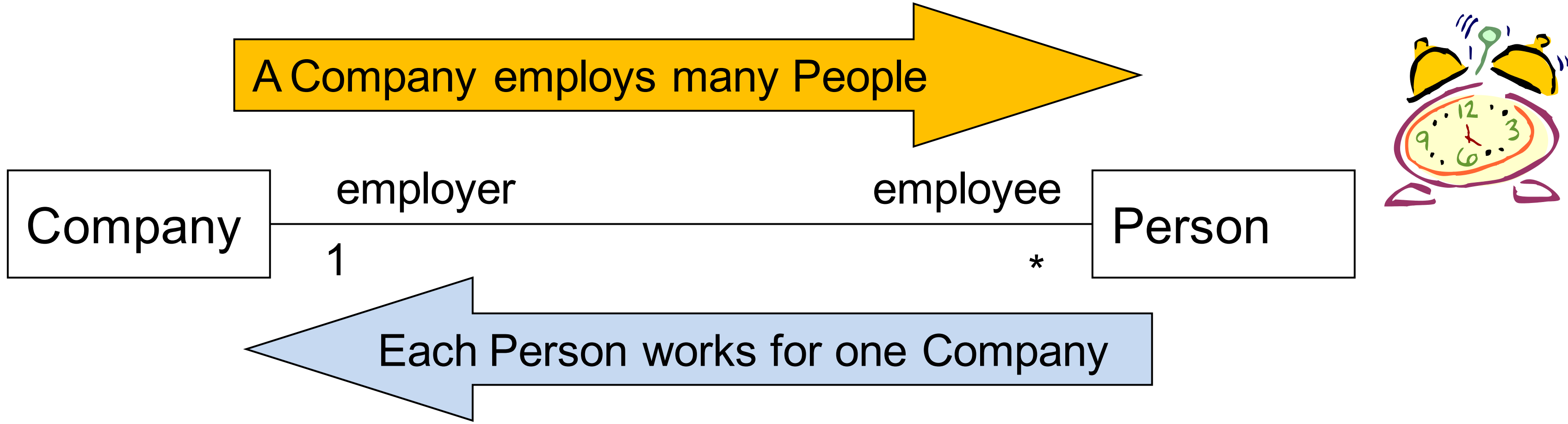




- An association can have role names *or* an association name
  - It's bad style to have both!
- The black triangle indicates the direction in which the association name is read:
  - "A Company employs many Persons"

Multiplicity

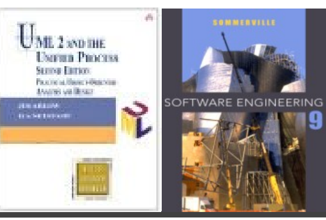




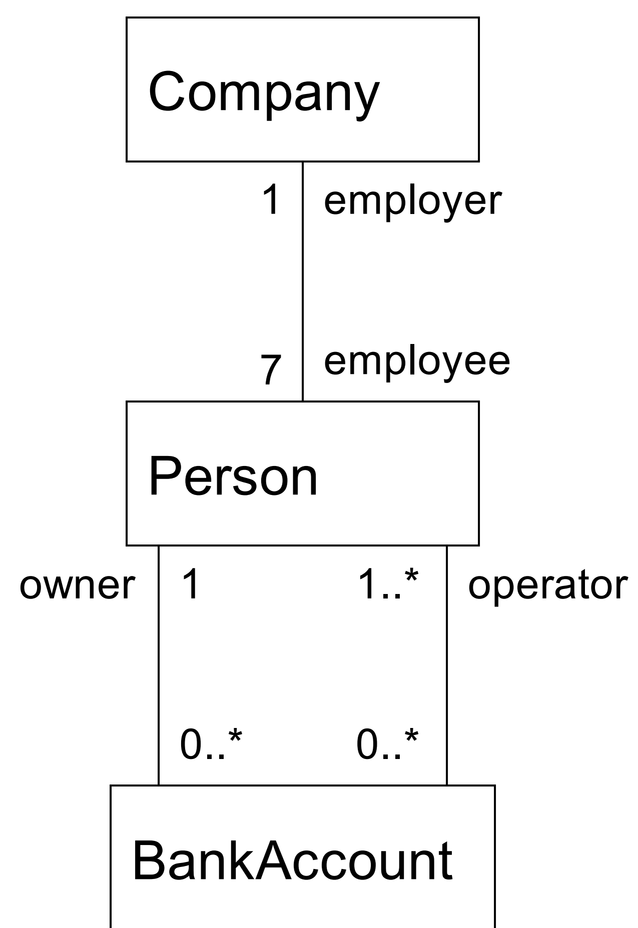
- Multiplicity is a constraint that specifies the number of objects that can participate in a relationship at *any point in time*
- If multiplicity is not explicitly stated in the model then it is undecided – *there is no default multiplicity*

multiplicity syntax: minimum..maximum	
0..1	zero or 1
1	exactly 1
0..*	zero or more
*	zero or more
1..*	1 or more
1..6	1 to 6

Multiplicity exercise

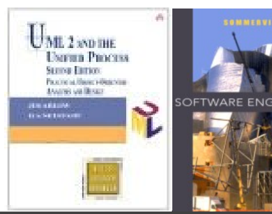


- How many
  - Employees can a Company have?
  - Employers can a Person have?
  - Owners can a BankAccount have?
  - Operators can a BankAccount have?
  - BankAccounts can a Person have?
  - BankAccounts can a Person operate?



© Clear View Training 2010 v2.6

Reflexive associations: file system example



subdirectory
0..*
Directory
1
0..*
File
0..1
parent

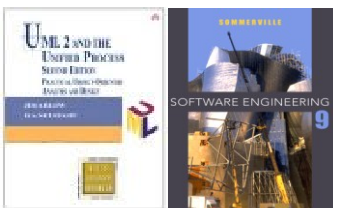
reflexive association

C
autoexec
config
Windows
My Documents
Corel
To John
Command

directories

files

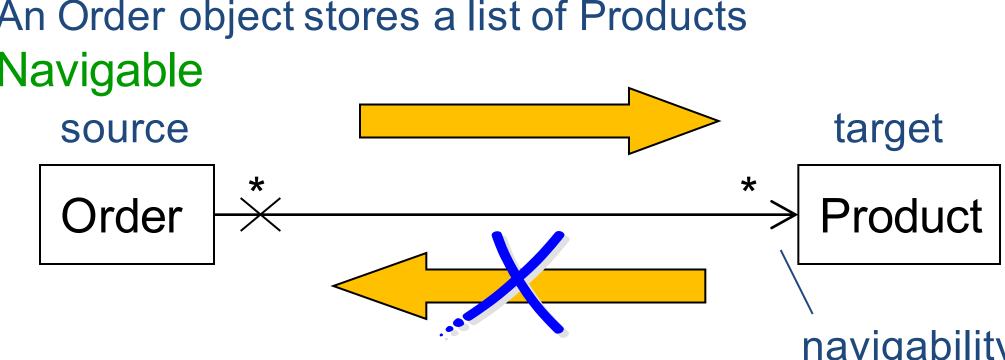
Hierarchies and networks

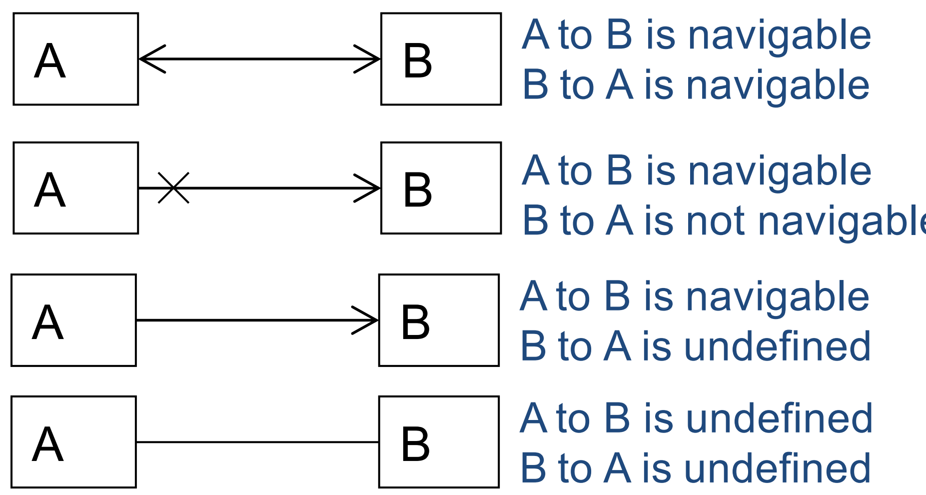


hierarchy	network
A 0..* 0..1	B 0..* 0..*
a1:A, b1:A, c1:A, d1:A, e1:A, f1:A, g1:A	c1:B, b1:B, f1:B, e1:B, d1:B, a1:B, g1:B
an an association hierarchy, each object has *zero or one* object directly above it	in an association network, each object has zero or many objects directly above it

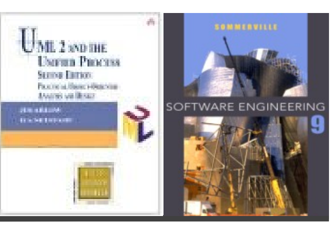
Navigability

- Navigability indicates that it is possible to traverse from an object of the *source* class to objects of the *target* class
  - Objects of the source class may reference objects of the target class using the role name
- Even if there is *no* navigability it might still be possible to traverse the relationship via some indirect means. However the computational cost of the traversal might be very high

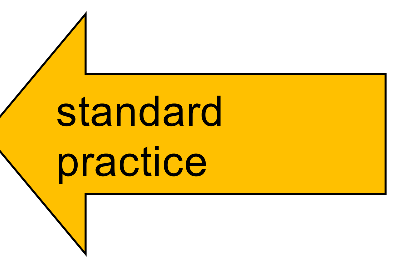


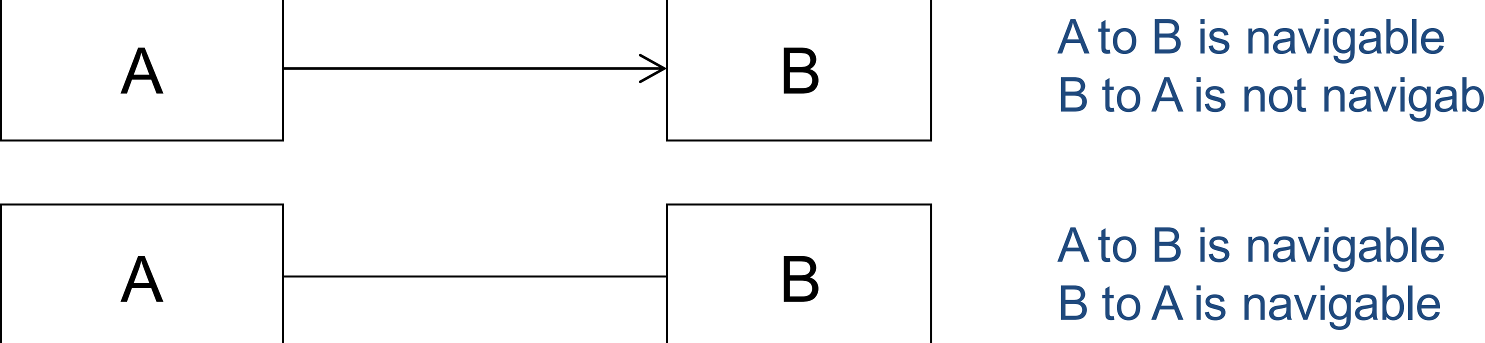


Navigability – standard practice



- Strict UML 2 navigability can clutter diagrams so the UML standard suggests three possible modeling idioms:
  1. Show navigability explicitly on diagrams with crosses and arrows
  2. Omit all navigability from diagrams
  3. Omit crosses from diagrams
     - bi-directional associations have no arrows
     - unidirectional associations have a single arrow
     - you can't show associations that are not navigable in either direction (not useful anyway!)

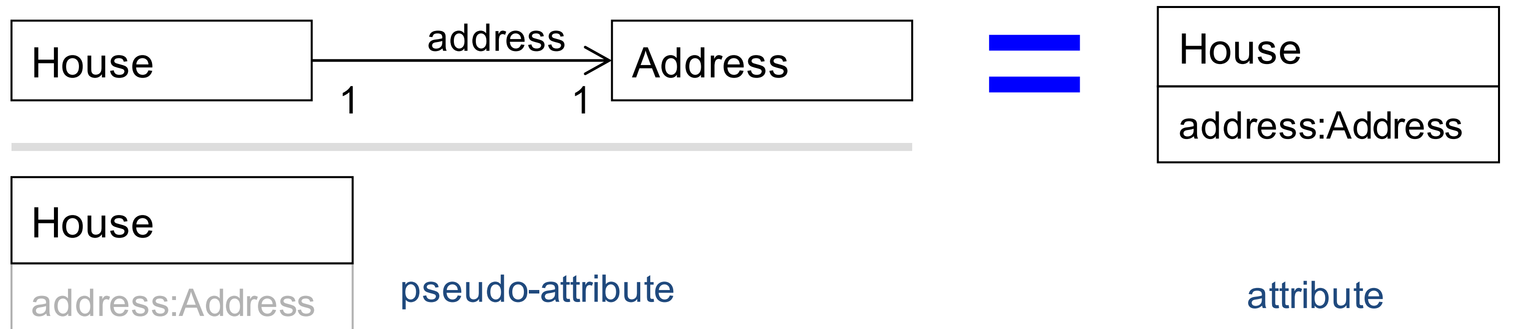




A to B is navigable
B to A is not navigable

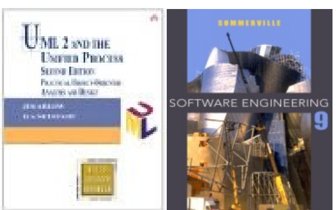
A to B is navigable
B to A is navigable

Associations and attributes



- If a navigable relationship has a role name, it is as though the source class has a pseudo-attribute whose attribute name is the role name and whose attribute type is the target class
- Objects of the source class can refer to objects of the target class using this pseudo-attribute
- Use associations when:
  - The target class is an important part of the model
  - The target class is a class that you have designed yourself and which must be shown on the model
- Use attributes when:
  - The target class is *not* an important part of the model e.g. a primitive type such as number, string etc.
  - The target class is just an implementation detail such as a bought-in component or a library component e.g. Java.util.Vector (from the Java standard libraries)

Association classes

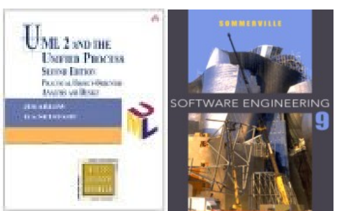


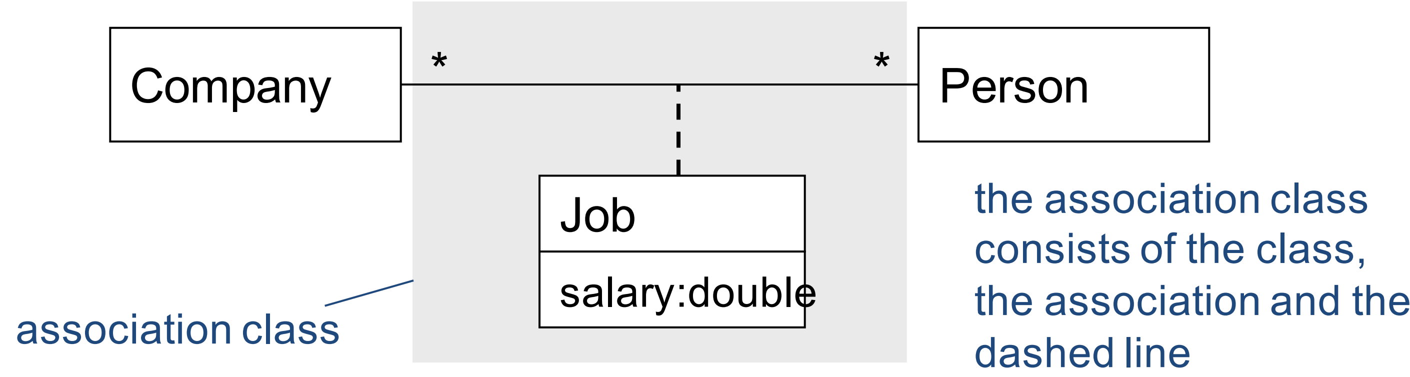
Each Person object can work for many Company objects.
Each Company object can employ many Person objects.
When a Person object is employed by a Company object, the Person has a salary.

But where do we record the Person's salary?

- Not on the Person class - there is a different salary for each employment
- Not on the Company class - different Person objects have different salaries
- The salary is a property of the employment relationship itself
  - every time a Person object is employed by a Company object, there is a salary

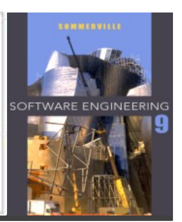
Association class syntax



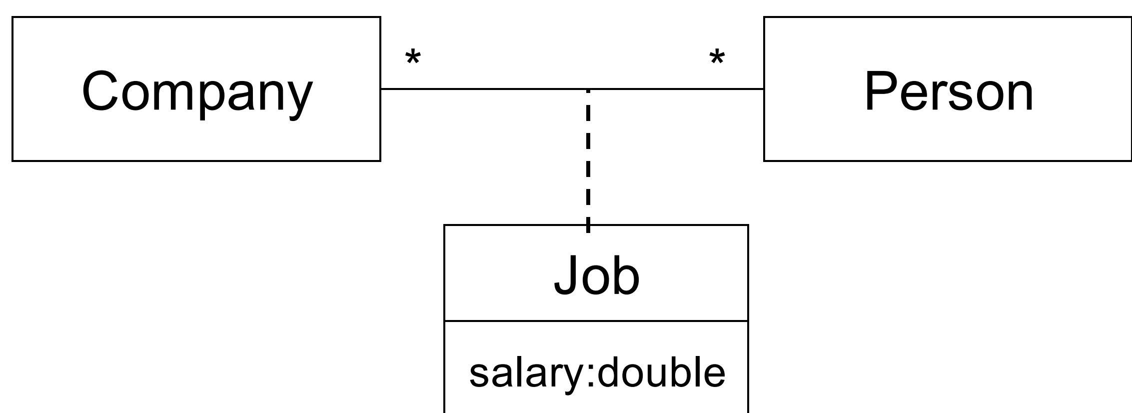


- We model the association itself as an association class. One instance of this class exists for each link between a Person object and a Company object
  - Instances of the association class are links that have attributes and operations
  - Can only use association classes when there is *one unique link* between two specific objects. This is because the identity of links is determined exclusively by the identities of the objects on the ends of the link
- We can place the salary and any other attributes or operations which are really features of the association into this class

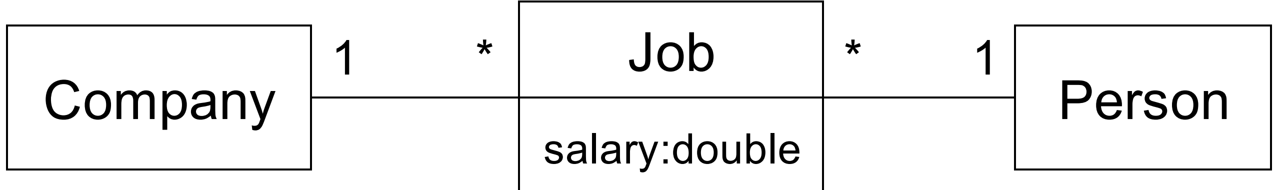
Using association classes



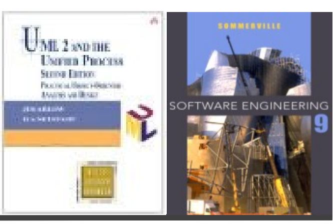
If we use an association class, then a particular Person can have only *one* Job with a particular Company



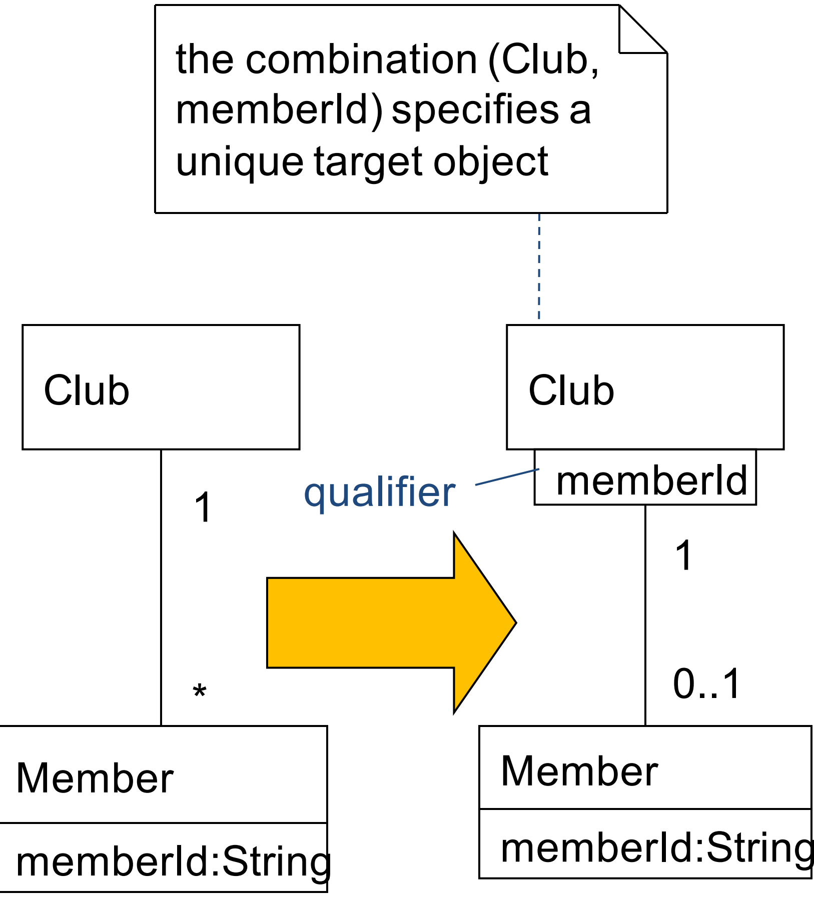
If, however a particular Person can have *multiple* jobs with the same Company, then we must use a *reified association*



Qualified associations



- Qualified associations reduce an n to many association to an n to 1 association by specifying a unique object (or group of objects) from the set
- They are useful to show how we can look up or navigate to specific objects
- Qualifiers usually refer to an attribute on the target class

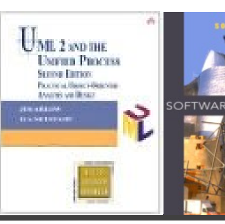


© Clear View Training 2010 v2.6

Dependencies

- "A dependency is a relationship between two elements where a change to one element (the supplier) may affect or supply information needed by the other element (the client)". In other words, the client *depends* in some way on the supplier
  - Dependency is really a catch-all that is used to model several different types of relationship. We've already seen one type of dependency, the «instantiate» relationship
- Three types of dependency:
  - Usage - the client uses some of the services made available by the supplier to implement its own behavior – this is the most commonly used type of dependency
  - Abstraction - a shift in the level of abstraction. The supplier is more abstract than the client
  - Permission - the supplier grants some sort of permission for the client to access its contents – this is a way for the supplier to control and limit access to its contents

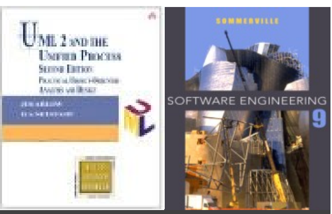
Usage dependencies



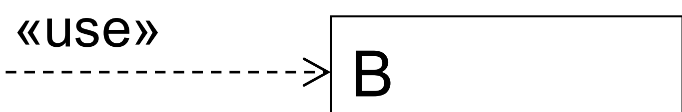
- «use» - the client makes use of the supplier to implement its behaviour
- «call» - the client operation invokes the supplier operation
- «parameter» - the supplier is a parameter of the client operation
- «send» - the client (an operation) sends the supplier (a signal) to some unspecified target
- «instantiate» - the client is an instance of the supplier

© Clear View Training 2010 v2.6

«use» - example



the stereotype is often omitted



A

foo(b : B)
bar() : B
doSomething()

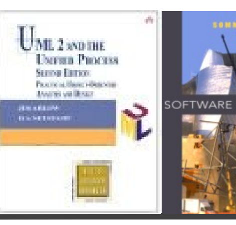
B

A «use» dependency is generated between class A and B when:

1) An operation of class A needs a parameter of class B
2) An operation of class A returns a value of class B
3) An operation of class A uses an object of class B somewhere in its implementation

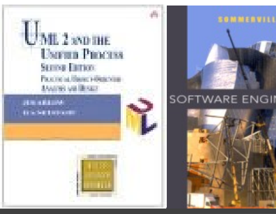
```
A :: doSomething()
{
  B myB = new B();
  …
}
```

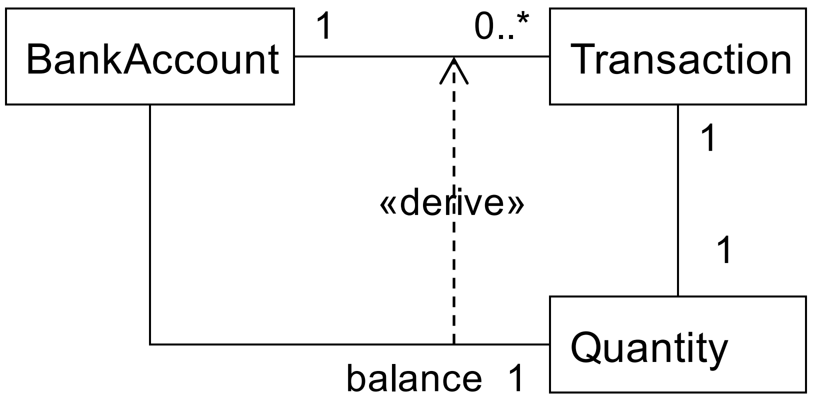
Abstraction dependencies

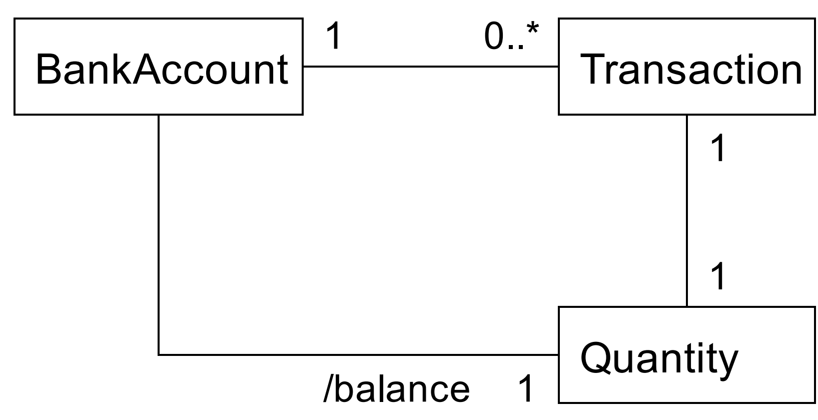


- «trace» - the client and the supplier represent the same concept but at different points in development
- «substitute» - the client may be substituted for the supplier at runtime. The client and supplier must realize a common contract. Use in environments that *don't* support specialization/generalization
- «refine» - the client represents a fuller specification of the supplier
- «derive» - the client may be derived from the supplier. The client is logically redundant, but may appear for implementation reasons

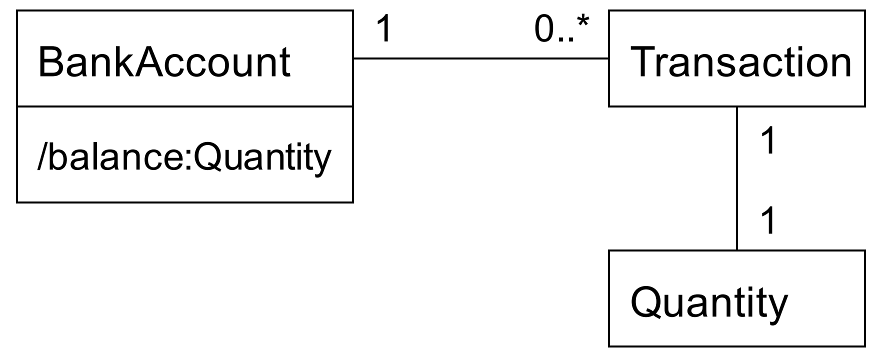
«derive» - example



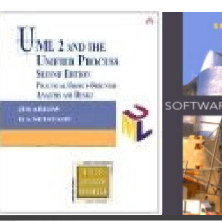




This example shows three possible ways to express a «derive» dependency



Permission dependencies



- The public contents of the supplier package are added as private elements to the namespace of the client package

✧ «import»

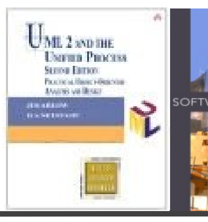
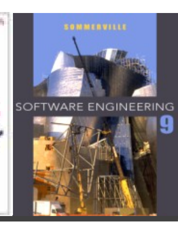
- The public contents of the supplier package are added as public elements to the namespace of the client package

✧ «permit»

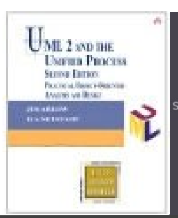
- The client element has access to the supplier element despite the declared visibility of the supplier

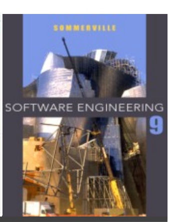
© Clear View Training 2010 v2.6

Key points

- ✧ Links – relationships between objects
- ✧ Associations – relationships between classes
  - role names
  - multiplicity
  - navigability
  - association classes
  - qualified associations
- ✧ Dependencies – relationships between model elements
  - usage
  - abstraction
  - permission



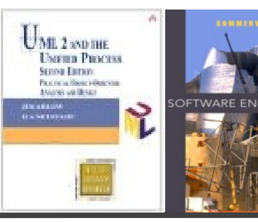


Inheritance and polymorphism

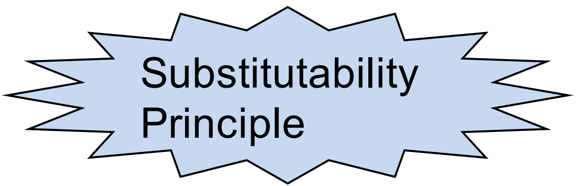
Lecture 5/Part 2

© Clear View Training 2010 v2.6

Generalisation

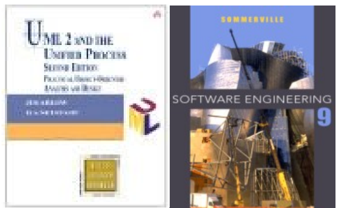


- A relationship between a more general element and a more specific element
- The more specific element is entirely consistent with the more general element but contains more information
- An instance of the more specific element may be used where an instance of the more general element is expected



© Clear View Training 2010 v2.6

Example: class generalisation



generalisation

specialisation

more general element

Shape

"is kind of"

Square

Circle

Triangle

more specific elements

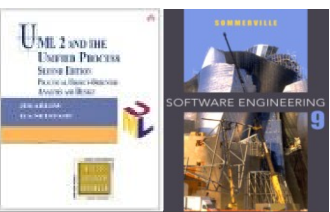
parent
superclass
base class
ancestor

child
subclass
descendent

A generalisation hierarchy

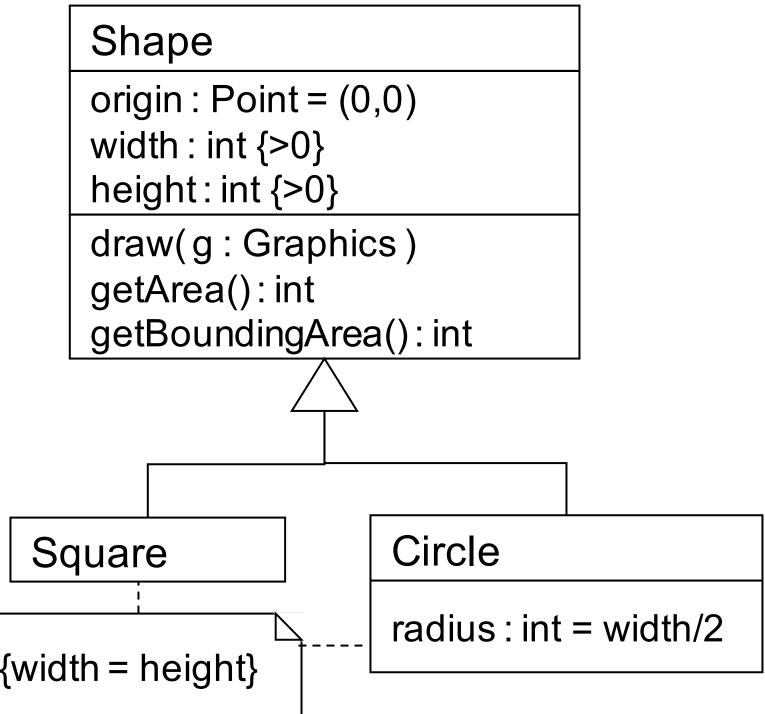
© Clear View Training 2010 v2.6

Class inheritance



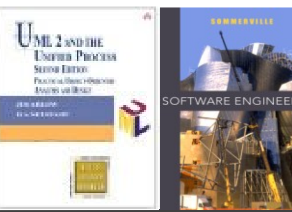
- Subclasses inherit *all* features of their superclasses:
  - attributes
  - operations
  - relationships
  - stereotypes, tags, constraints
- Subclasses can add new features
- Subclasses can override superclass operations
- We can use a subclass instance *anywhere* a superclass instance is expected

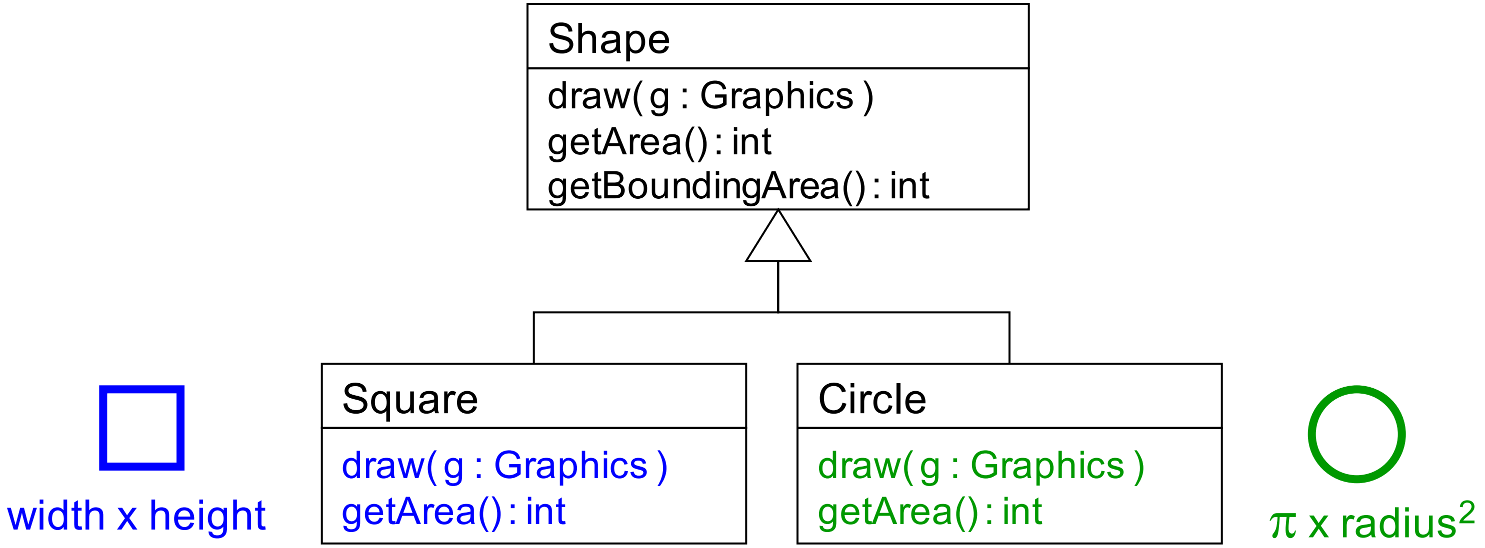
Substitutability Principle



But what's wrong with these subclasses

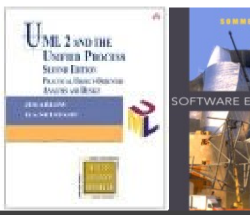
Overriding





- Subclasses often need to *override* superclass behaviour
- To override a superclass operation, a subclass must provide an operation with the same signature
  - The operation signature is the operation name, return type and types of all the parameters
  - The names of the parameters don't count as part of the signature

Abstract operations & classes



abstract class

Shape
draw(g : Graphics) *getArea() : int* getBoundingArea() : int

abstract operations

abstract class and operation names *must* be in italics

concrete operations

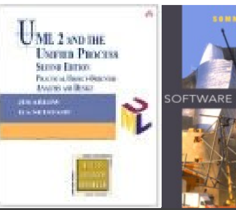
concrete classes

Square
draw(g : Graphics) getArea() : int

Circle
draw(g : Graphics) getArea() : int

- We can't provide an implementation for
  Shape :: draw(g : Graphics) or for
  Shape :: getArea() : int
  because we don't know how to draw or calculate the area for a "shape"!
- Operations that lack an implementation are *abstract operations*
- A class with any abstract operations *can't* be instantiated and is therefore an *abstract class*

Exercise



what's wrong
with this model?

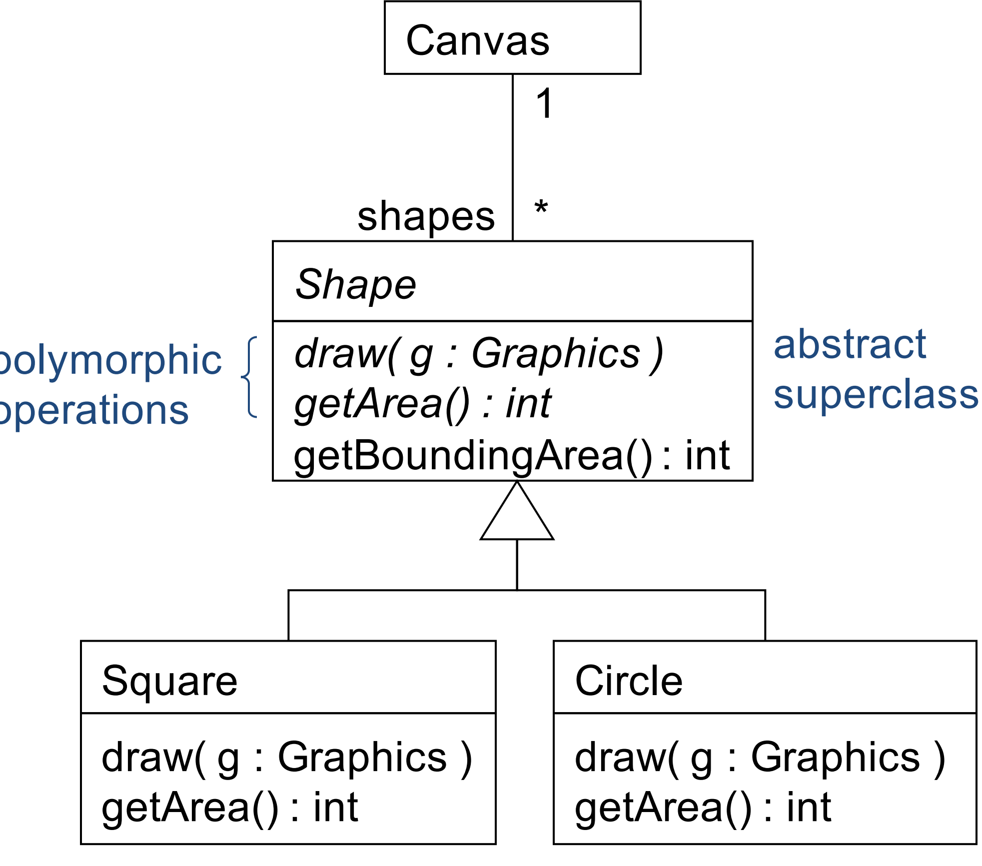
Vehicle

JaguarXJS

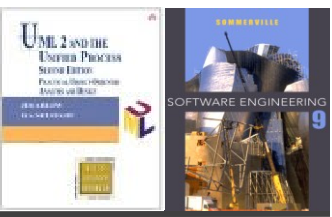
Truck

Polymorphism

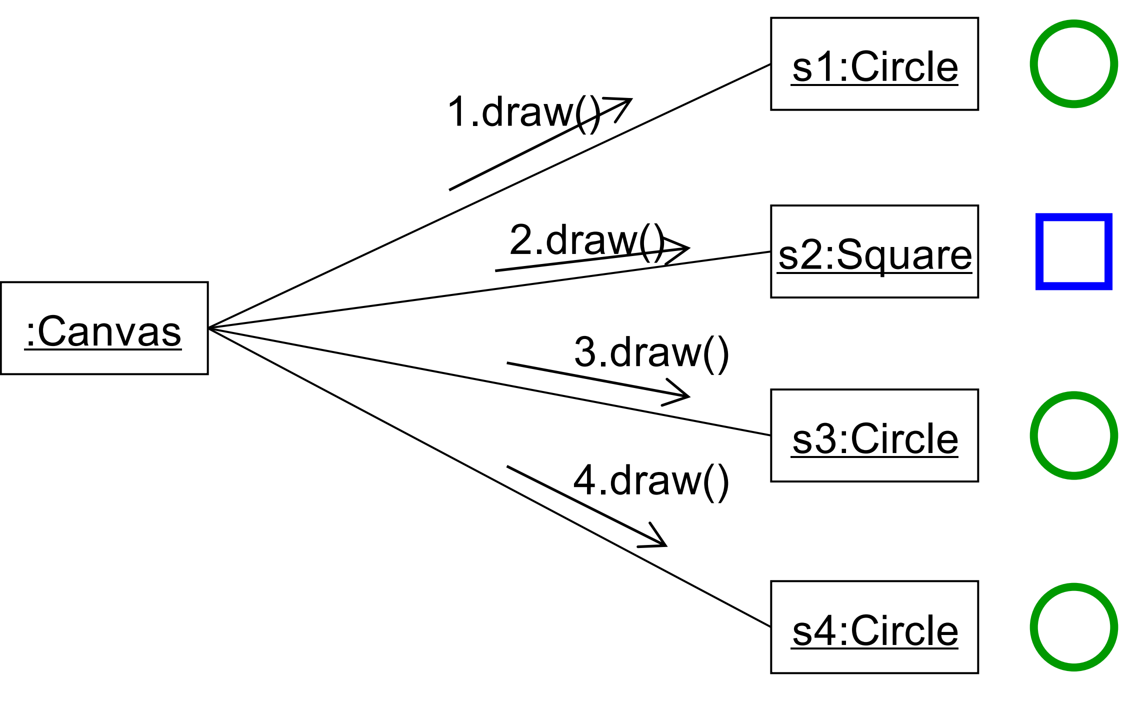
- Polymorphism = "many forms"
  - A polymorphic operation has many implementations
  - Square and Circle provide implementations for the polymorphic operations *Shape::draw()* and *Shape::getArea()*
- All concrete subclasses of Shape *must* provide concrete draw() and getArea() operations because they are abstract in the superclass
  - For draw() and getArea() we can treat all subclasses of Shape in a similar way - we have defined a contract for Shape subclasses



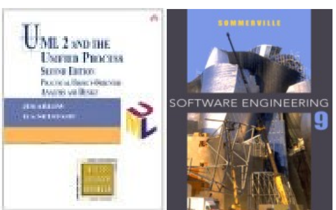
What happens?



- Each class of object has its own implementation of the draw() operation
- On receipt of the draw() message, each object invokes the draw() operation specified by its class
- We can say that each object "decides" how to interpret the draw() message based on its class



BankAccount example



Bank

1

*

BankAccount

withdraw()
calculateInterest()
deposit()

ShareAccount

withdraw()
calculateInterest()
deposit()

CheckingAccount

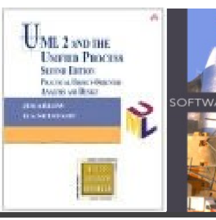
withdraw()
calculateInterest()

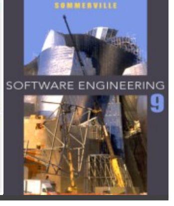
DepositAccount

withdraw()
calculateInterest()

- We have overridden the deposit() operation even though it is *not* abstract. This is perfectly legal, and quite common, although it is generally considered to be bad style and should be avoided if possible

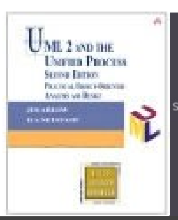
Key points

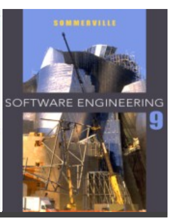




- Subclasses:
  - inherit *all* features from their parents including constraints and relationships
  - may add *new* features, constraints and relationships
  - may *override* superclass operations
- A class that can't be instantiated is an abstract class

© Clear View Training 2010 v2.6



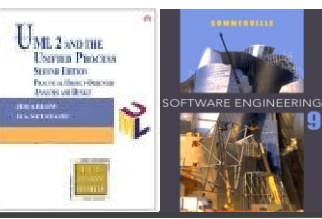


Interaction Diagrams

Lecture 5/Part 3

© Clear View Training 2010 v2.6

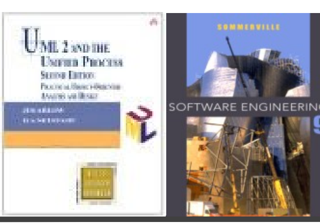
Use Case realization



✧ Use case realizations consist of the following elements:

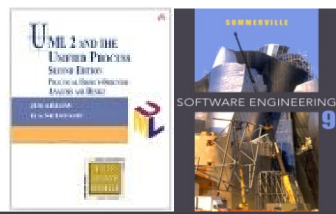
- Analysis class diagrams
  - These show relationships between the analysis classes that interact to realise the UC
- Interaction diagrams
  - These show collaborations between specific objects that realise the UC. They are “snapshots” of the running system
- Special requirements
  - UC realization may well uncover new requirements specific to the use case. These must be captured
- Use case refinement
  - We may discover new information during realization that means that we have to update the original UC

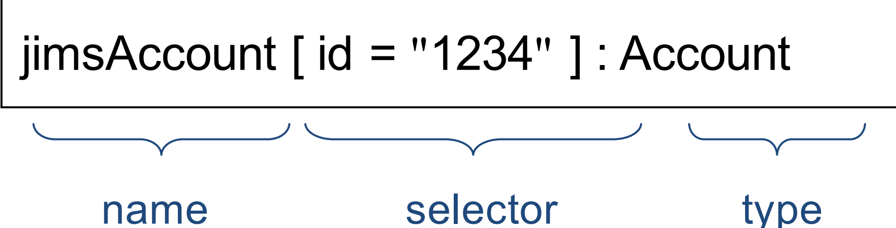
Interactions



- ✧ Interactions are units of behavior of a context classifier
- ✧ In use case realization, the context classifier is a use case
  - The interaction shows how the behavior specified by the use case is realized by instances of classifiers
- ✧ Interaction diagrams capture an interaction as:
  - Lifelines – participants in the interaction
  - Messages – communications between lifelines

Lifelines

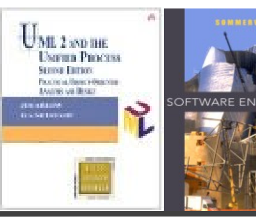




- A lifeline represents a single participant in an interaction
  - Shows how a classifier instance may participate in the interaction
- Lifelines have:
  - name - the name used to refer to the lifeline in the interaction
  - selector - a boolean condition that selects a specific instance
  - type - the classifier that the lifeline represents an instance of
- They *must* be uniquely identifiable within an interaction by name, type or both
- The lifeline has the same icon as the classifier that it represents
  - The lifeline jimsAccount represents an instance of the Account class
  - The selector [id = "1234"] selects a specific Account instance with the id "1234"

© Clear View Training 2010 v2.6

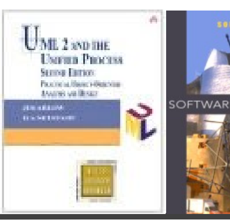
Messages



✧ A message represents a communication between two lifelines

sender ⇨ receiver/ target	type of message	semantics
——▶	synchronous message	calling an operation synchronously the sender waits for the receiver to complete
——>	asynchronous send	calling an operation asynchronously, sending a signal the sender *does not* wait for the receiver to complete
<- - - -	message return	returning from a synchronous operation call the receiver returns focus of control to the sender
- - - -> :A	creation	the sender creates the target
——> X	destruction	the sender destroys the receiver
●——>	found message	the message is sent from outside the scope of the interaction
——>●	lost message	the message fails to reach its destination

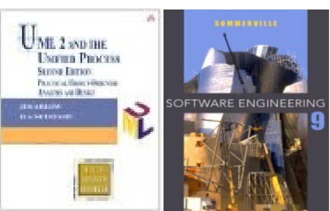
Interaction diagrams



- Sequence diagrams
  - Emphasize time-ordered sequence of message sends
  - Show interactions arranged in a time sequence
  - Are the richest and most expressive interaction diagram
  - Do not show object relationships explicitly - these can be inferred from message sends
- Communication diagrams
  - Emphasize the structural relationships between lifelines
  - Use communication diagrams to make object relationships explicit
- Interaction overview diagrams
  - Show how complex behavior is realized by a set of simpler interactions (discussed earlier together with Activity diagrams)
- Timing diagrams
  - Emphasize the real-time aspects of an interaction

© Clear View Training 2010 v2.6

Sequence diagram syntax



sd AddCourse

:Registrar

:RegistrationManager

uml:Course

addCourse("UML")

«create»

The Registrar selects "add course".

The system creates the new Course.

synchronous message

lifeline

object creation message

activation

message return

notes can form a "script" describing the flow

object is created at this point

- *All* interaction diagrams may be prefixed sd to indicate their type
  - You can generally infer diagram types from diagram syntax
- Activations indicate when a lifeline has focus of control - they are often omitted from sequence diagrams

Deletion and self-delegation

sd DeleteCourse

:Registrar

:RegistrationManager

uml:Course

deleteCourse("UML")

self delegation

findCourse("UML")

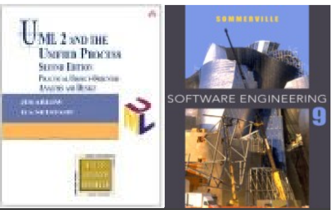
nested activation

«destroy»

object is deleted at this point

- Self delegation is when a lifeline sends a message to itself
  - Generates a nested activation
- Object deletion is shown by terminating the lifeline's tail at the point of deletion by a large X

State invariants and constraints



sd ProcessAnOrder

:Customer

:OrderManager

:DeliveryManager

raiseOrder()

«create»

:Order

state invariant

unpaid

label

A

acceptPayment()

acceptPayment()

paid

constraint

deliver()

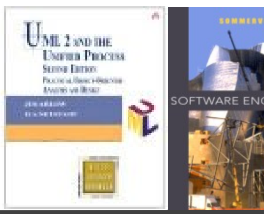
{B – A <= 28 days}

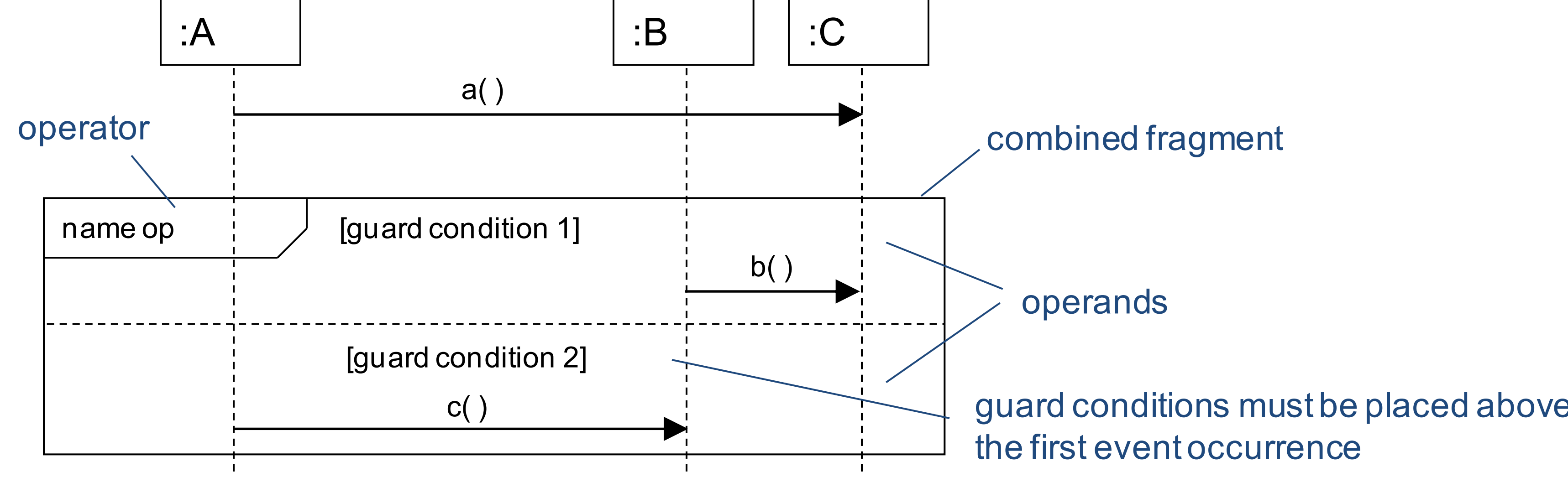
deliver()

delivered

B

Combined fragments





- Sequence diagrams may be divided into areas called *combined fragments*
- Combined fragments have one or more *operands*
- *Operators* determine how the operands are executed
- *Guard conditions* determine whether operands execute. Execution occurs if the guard condition evaluates to true
  - A single condition may apply to all operands OR
  - Each operand may be protected by its own condition

Common operators

operator	long name	semantics
opt	Option	There is a single operand that executes if the condition is true (like if … then)
alt	Alternatives	The operand whose condition is true is executed. The keyword else may be used in place of a Boolean expression (like select… case)
loop	Loop	This has a special syntax: loop min, max [condition] Iterate min times and then up to max times while condition is true
break	Break	The combined fragment is executed rather than the rest of the enclosing interaction
ref	Reference	The combined fragment refers to another interaction

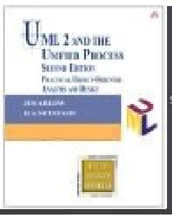
ref

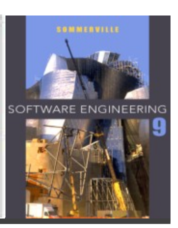
findStudent(name):Student

ref has a single operand that is a reference to another interaction.

This is an *interaction use.*

The rest of the operators

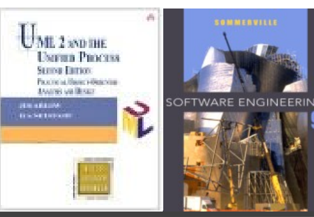




- These operators are less common

operator	long name	semantics
par	parallel	Both operands execute in parallel
seq	weak sequencing	The operands execute in parallel subject to the constraint that event occurrences on the *same* lifeline from *different* operands must happen in the same sequence as the operands
strict	strict sequencing	The operands execute in strict sequence
neg	negative	The combined fragment represents interactions that are invalid
critical	critical region	The interaction must execute atomically without interruption
ignore	ignore	Specifies that some messages are intentionally ignored in the interaction
consider	consider	Lists the messages that are considered in the interaction (all others are ignored)
assert	assertion	The operands of the combined fragments are the only valid continuations of the interaction

Branching with opt and alt



- opt semantics:
  - single operand that executes if the condition is true
- alt semantics:
  - two or more operands each protected by its own condition
  - an operand executes if its condition is true
  - use else to indicate the operand that executes if *none* of the conditions are true

sd example of opt and alt

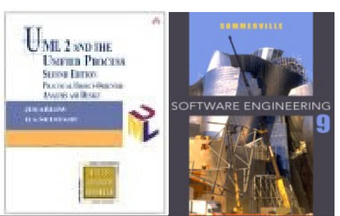
:A
:B
:C
:D

opt [condition]
do this if condition is true

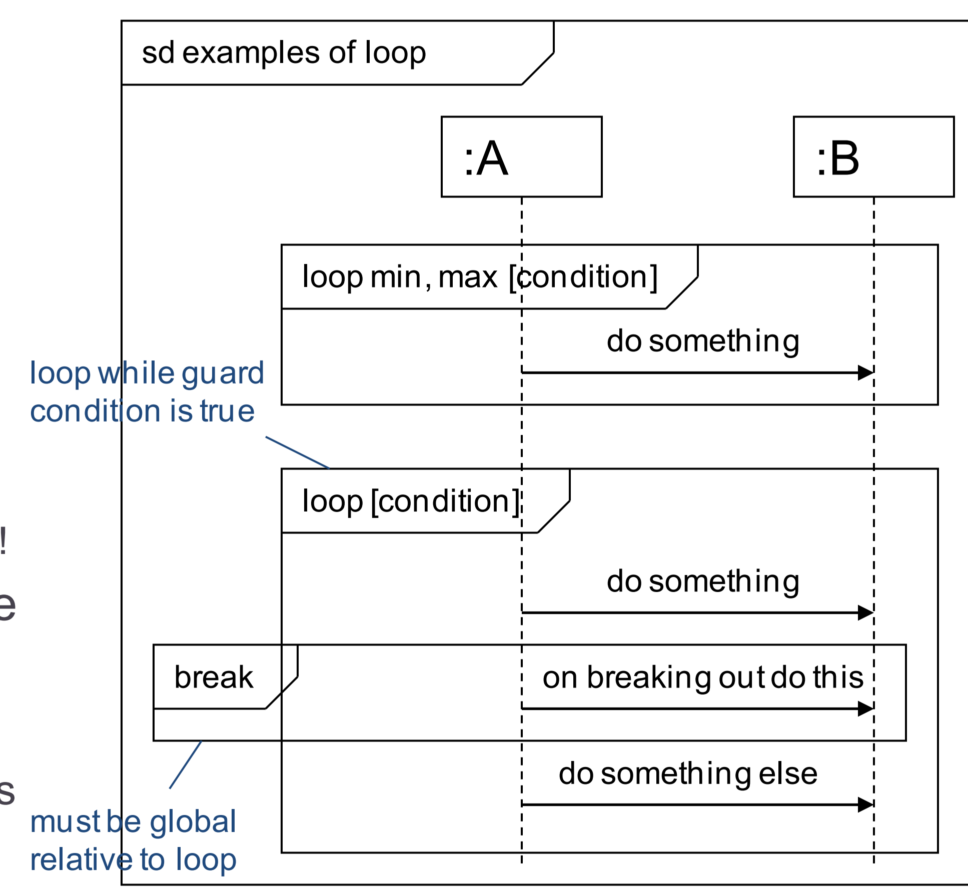
alt
[condition1]
do this if condition1 is true
[condition2]
do this if condition2 is true
[else]
do this if neither condition is true

© Clear View Training 2010 v2.6

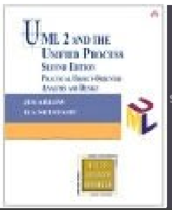
Iteration with loop and break



- loop semantics:
  - Loop min times, then loop (max – min) times while condition is true
- loop syntax
  - A loop without min, max or condition is an infinite loop
  - If only min is specified then max = min
  - condition can be
    - Boolean expression
    - Plain text expression *provided* it is clear!
- Break specifies what happens when the loop is broken out of:
  - The break fragment executes
  - The rest of the loop after the break does *not* execute
- The break fragment is *outside* the loop and so should overlap it as shown



Loop idioms



type of loop	semantics	loop expression
infinite loop	keep looping forever	loop *
for i = 1 to n {body}	repeat (n) times	loop n
while(booleanExpression) {body}	repeat while booleanExpression is true	loop [booleanExpression]

repeat
{body}
while(booleanExpression)
	execute once then repeat while booleanExpression is true	loop 1, * [booleanExpression]

forEach object in collection {body}	Execute the loop once for each object in a collection	loop [for each object in collection]
forEach object in ObjectType {body}	Execute the loop once for each object of a particular type	loop [for each object in :ObjectType]

© Clear View Training 2010 v2.6

Sequence diagrams in design

sd AddCourse - design

:Registrar

:RegistrationUI

:RegistrationManager

:DBManager

addCourse("UML")

addCourse("UML")

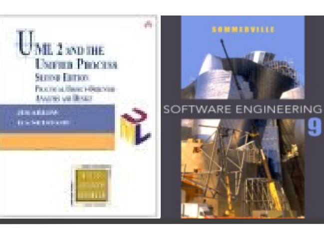
uml = Course("UML")

uml:Course

save(uml)

© Clear View Training 2010 v2.6

Communication diagram syntax



- ✧ Communication diagrams emphasize the structural aspects of an interaction - how lifelines connect together
  - Compared to sequence diagrams they are semantically weak
  - Object diagrams are a special case of communication diagrams

sd AddCourses

sequence number

message

lifeline

uml:Course

1.1: «create»

1: addCourse("UML")

2: addCourse("MDA")

:RegistrationManager

:Registrar

link

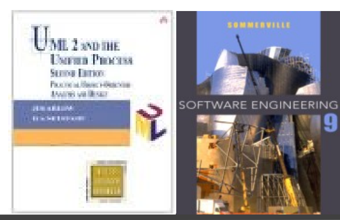
2.1: «create»

mda:Course

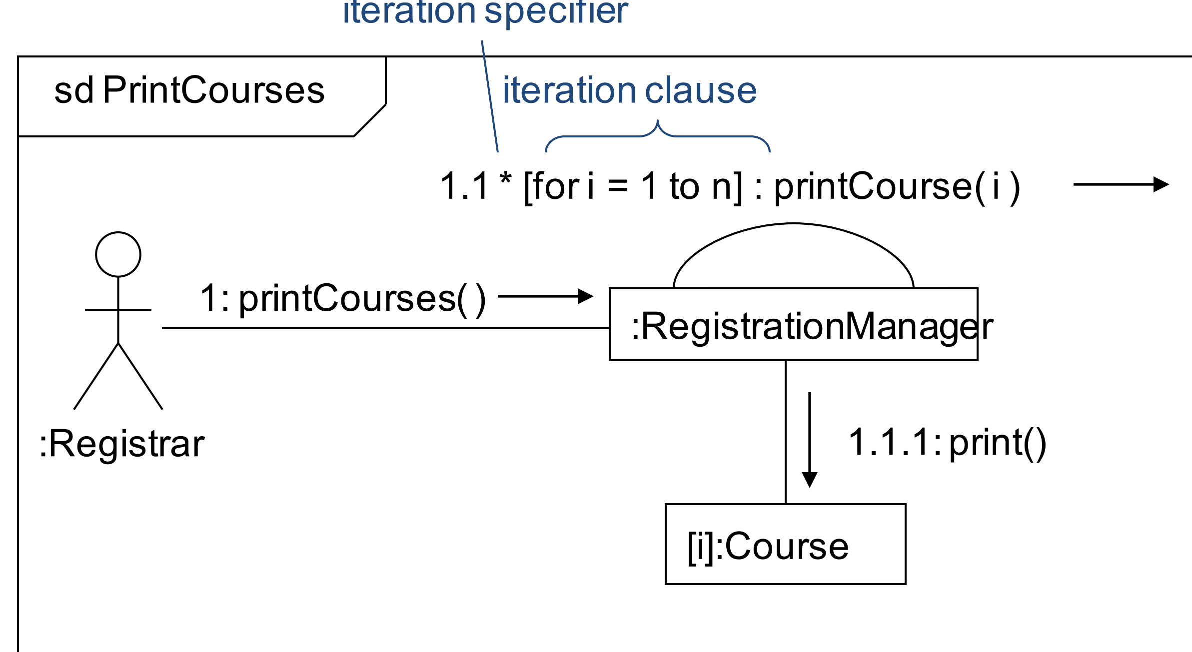
object creation message

© Clear View Training 2010 v2.6

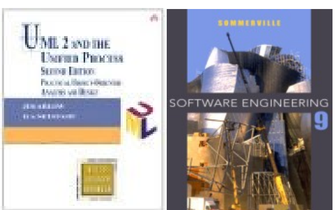
Iteration



- Iteration is shown by using the *iteration specifier* (*), and an optional *iteration clause*
  - There is no prescribed UML syntax for iteration clauses
  - Use code or pseudo code
- To show that messages are sent in parallel use the parallel iteration specifier, *//



Branching



return value from message

sd register student for course

1.1: student = findStudent("Jim")
1.2: course = findCourse("UML")

1: register ("Jim", "UML")

:RegistrationManager

1.4 [!found] : error()

:Registrar

1.3 [found] : register(student)

guard condition

found = (student != null) & (course != null)

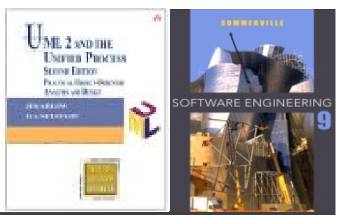
course:Course

It's hard
to show
branching
clearly!!

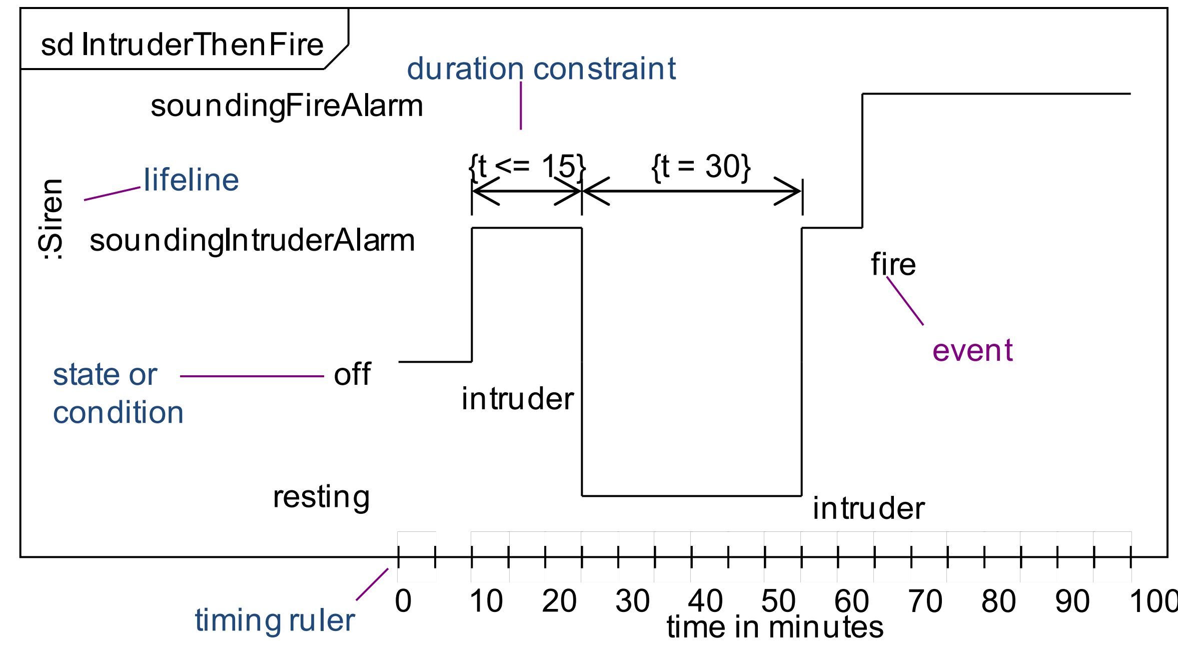
- Branching is modelled by prefixing the sequence number with a *guard condition*
  - There is no prescribed UML syntax for guard conditions!
  - In the example above, we use the variable found. This is true if both the student and the course are found, otherwise it is false

© Clear View Training 2010 v2.6

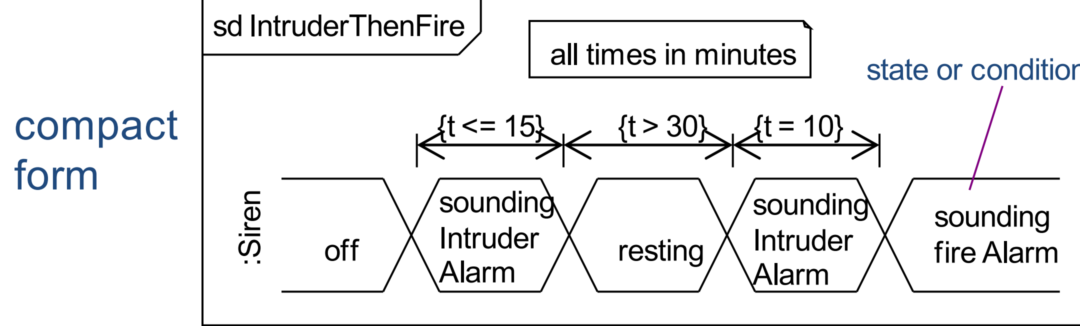
Timing diagrams



- Emphasize the real-time aspects of an interaction
- Used to model timing constraints
- Lifelines, their states or conditions are drawn vertically, time horizontally
- It's important to state the time units you use in the timing diagram



compact form



Messages on timing diagrams

- You can show messages between lifelines on timing diagrams
- Each lifeline has its own partition

sd SirenBehavior

all times in minutes

{t <= 0.016}

:FireSensorMonitor

triggered

notTriggered

:IntruderSensorMonitor

{t <= 0.016}

triggered

notTriggered

messages

soundFireAlarm()

:Siren

soundingFireAlarm

soundingIntruderAlarm

off

resting

soundIntruderAlarm()

soundIntruderAlarm()

soundIntruderAlarm()

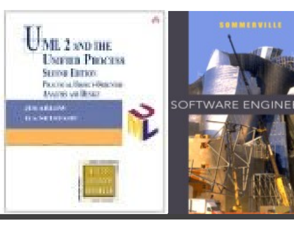
soundIntruderAlarm()

{t <= 15}

{t = 30}

{t <= 15}

Key points



- In this section we have looked at use case realization using interaction diagrams
- There are four types of interaction diagram:
  - Sequence diagrams – emphasize time-ordered sequence of message sends
  - Communication diagrams – emphasize the structural relationships between lifelines
  - Interaction overview diagrams – show how complex behavior is realized by a set of simpler interactions; presented together with Activity diagrams
  - Timing diagrams – emphasize the real-time aspects of an interaction

© Clear View Training 2010 v2.6